React JS

What is react?

* It’s a library not a framework
* Focus only on UI, don’t focus on http calls/routings, only responsible for rich Ui
* Has rich ecosystem for other purposes.
* Its component base architecture.
* Can write reusable code, that component can be used in angular or vue.

Declarative Paradigm: just tell what to do, how it does is up to them.

Imperative paradigm: we explicitly say what to done step to step, we control the flow.

* React is declarative (tell what you want it build actual UI). Declarative paradigm in contrast to imperative paradigm we use to which implements algorithm explicit steps.
* We have to what to create react will create actual UI.
* It handles efficiently and updates and renders the exact component and updates the DOM.
* It can be integrated to any app as a portion or complete page or whole application.
* React native is used for mobile apps.
* Html, CSS, JavaScript, ES6,
* In JavaScript – ‘this’ keyword, filter, map and reduce.
* In ES6 – let & const, arrow functions, template literals, default parameters, object literals, rest and spread operators and DE-structuring assignment.

Folder structure

* We want react to control the app for that we have one div tag with id = root one time react app will take control and it responsible rendering. (this is called as root DOM node)
* This root is tagged to index.js file with that elementById(‘root’).

Component structure:

* Components can contain many components ex: app component (acts as parent component)
* In react we have two types of components:

1. Stateless Functional component
2. Stateful class component

|  |  |
| --- | --- |
| Stateless functional component | Stateful class component |
| 1. Are literally JS functions they return html which describe the UI. | Class extending component class.  Render method returning HTML |

Functional vs Class Components

|  |  |
| --- | --- |
| Functional component | Class component |
| 1. Simple functions receiving props and returning declaration. 2. Use functional component as much as possible. 3. Absence of ‘this’ keyword. 4. Solution without using state 5. Mainly responsible for the UI 6. Also called as Stateless/Dumb/Presentational | 1. More feature rich 2. Maintain their own private data – state 3. Complex UI logic 4. Provide lifecycle hooks 5. Stateful/Smart/Container components |

**JSX (JavaScript XML)**

Write XML-like code for elements and components.

JSX tags have a tag name, attributes and children.

JSX makes code simpler and elegant. (Not necessity to use JSX)

JSX ultimately transpiles to pure JS which is understood by the browser.

**JSX differences**

Class -> className

For -> htmlFor

CamelCase property

* onclick -> onClick
* tabindex -> tabIndex

**Props**: are immutable

React components should work as pure function with respect to props parameter cannot change at any time.

All the application make use of props.

**Props VS State**

|  |  |
| --- | --- |
| Functional component | Class component |
| 1. props get passed to the components 2. Functional parameters 3. props are immutable 4. Accessed as props – Functional Components 5. Accessed as useState Hook – Functional Components | 1. State is managed within the component 2. Variables declared with in the function body 3. It is managed within the component so it has full control 4. Accessed as this.props – Class Components 5. Accessed as this.state – Class Components |

**Do’s and DoNot’s**

* We should never modify **state** directly. Use **setState**.
* Whenever you want to execute code after state has changed don’t place codes after **setState** method instead place that in callback function, and pass it as a second parameter for **setState** method.
* **this**.setState(

{count: **this**.state.count + 1},  
 () => {console.log('from call back', **this**.state.count)}

);

* React group multiple **setState** call to single update for better performance.
* Whenever you want to update state base on the previous state make sure to pass function as an argument instead of passing regular object.
* **this**.setState((prevState, props) => ({  
   count: prevState.count + props.index  
  }), () => {  
   console.log('from callback', **this**.state.count)  
  });
* Only place we can assign state is constructor, any other place if you want to update state you have to use **setState** method.
* Call to **setState** are asynchronies, if you place next to setState it will not update.

**Destructuring Props and State:**

* Destructuring in the parameter.
* **const** Greet = ({name, greet, children}) => {  
   **return** (<div><h1>Hello {name} {greet}</h1>{children}</div>)  
  };
* Destructuring in function body.
* **const** Greet = props => {  
   **const** {name, greet, children} = props;  
   **return** (<div><h1>Hello {name} {greet}</h1>{children}</div>)  
  };
* You can extract necessary props form the props object instead of all.

**Event Handling**

* Ex: onClick={clickHandler}
* We should not use as a function call Ex: onClick={clickHandler()} this will get trigged when the component get rendered. This comes worst in class component.
* In class component Ex: onClick={this.clickHandler}

**Bind Events**

1. Binding in the render method. Ex: onClick={this.clickHandler.bind(this)} this is the default functionality of JS by using bind(this).
2. Arrow function approach in render method, using Ex: onClick={() => this.clickHandler()} in this () are required. \*\*If you want to pass parameters use this approach.
3. Binding in the constructor, this the official approach in react docs. This is better than render method. Ex: this.clickHandler = this.clickHandler.bind(this)
4. Use arrow function as class property it’s the way you define the method.

**Parent Child Components Communication** (methods as props)

* We use props to pass any data from parent to child component.
* To communicate child to parent we still use props but pass in reference to method as props to the child component.
* In the child component use props object to access the method in parent component.
* If you want to pass parameter form the child to parent use arrow functions.

**Conditional Rendering**  Ex: UserGreeting.js

1. If/else

* If/else do not work inside **JSX** because JSX is a syntax sugar for function calls and object construction.

Ex: **if** (**this**.state.isLoggedIn) {**return** <div>Welcome srini</div>}  
**else** {**return** <div>Welcome Guest</div>}

1. Element variables

* Using temporary variable and return the value.

**Ex: let** message;  
**if** (**this**.state.isLoggedIn) {message = <div>Welcome srini</div>}   
**else** {message = <div>Welcome Guest</div>}  
**return** <div>{message}</div>

1. Ternary conditional operator

Ex: **this**.state.isLoggedIn ? <div>Welcome srini</div> : <div>Welcome Guest</div>

1. Short circuit operator

Ex: **return this**.state.isLoggedIn && <div>Welcome srini</div>

**List Rendering**

* Rendering list of data using map() as we in JavaScript.

Ex: **const** names = [ 'sri', 'srini', 'srinivas'];  
 **const** nameList = names.map(name => <h2>{name}</h2>);  
 **return**(<div>{nameList}</div>)

* Simplify as much as possible. Ex: NameList.js and Person.js
* **Key** is a special string attribute you need to include when creating list of elements.
* **Keys** give the elements a stable identity.
* **Keys** help react to identify which items have changed or added or removed to help the efficient update the **UI.**
* Each item render in the list by using map must have a **prop** called **key** and the **value** the **prop** must be **unique** with in the **list.**
* **Key** prop is **reserved** not accessible in the child component it’s only used to render the list efficiently, if you want the access same value within the child component you should pass it as a different prop.
* While there is an update in the lists React renders both at the same time and generates mutation whenever there is difference it simply **insert** the item in to the **DOM** tree instead for clearing old tree and construct new tree.
* Without **key** props React end up thinking the entire list items are different this is **inefficient** way of doing and reduce the **performance**.

**Index as Key**

* When to use **index** as a **key?**
  1. The items in the list do not have a unique id.
  2. The list is a static list and will not change.
  3. The list will never be reordered or filtered.
* To avoid using **index** as **key**
  1. Try one of **npm** packages or hashing one the unique value from one the existing properties
* **Try to avoid using index as key.** In fact react uses index as a key if we not specify the key.

**Styling React Components**

1. CSS style sheets

**In js file**

**function** Stylesheet(props) {  
 **let** className = props.primary ? 'primary' : '';  
 **return**(<div><h1 className={`${className} size-xl`}>Stylesheet</h1></div>)  
}

**In CSS file**

.primary{color: orange;}  
.size-xl{font-size: 50px;}

1. Inline styling

**const** heading ={fontSize: '40px', color: 'blue'};  
**function** Inline() {  
 **return**(<div><h1 style={heading}>Inline</h1></div>)  
}

1. CSS Modules

* This are from CSS modules, the file name Ex: appStyles.module.css
* This is the best approach of using CSS because we reference styles to a local variable it can’t be used in the children component.

In js file

* import styles from './appStyles.module.css';
* <h1 className={styles.success}>success</h1>

In CSS file

* .success {color: green;}

1. CSS in JS Libaries

**Form Handling**

1. Form elements whose values are controlled by react is called **controlled component.**
2. First initial value is set from the state and propagating changed value to state and back to field, react always have the access to component state which reflects the updated values of the form elements that state object can be used to submit the form data when needed.
3. Whenever there is a change in **onChange={}** method is triggered and updates the state in a cyclic process.
4. To prevent reloading the page and emptying the form we have to use **preventDefault()** method.

.preventDefault()

1. It always better to use **onSubmit={this.handleSubmit}** in the **form** button type=”submit”.

**Mounting Lifecycle Methods**

1. **Constructor (props)**

* This is call whenever new component is created.
* This is perfect for initializing state and binding the event handlers.
* Never **call** **HTTP requests** form constructor it causes side effects like making ajax calls.
* We have to call special function **super(props)** this will call base class constructor.
* We have access to **props** only after calling **super()** after passing props as an argument.
* This only place you can set or change the state directly by overwriting **this.state** fields in all other cases you to make use of **this.setState().**

1. **Static getDerivedStateFromProps(props, State)**

* This is very rarely used lifecycle method.
* This many used when the state depends on the changes in props over time.
* Ex: we have a component but the initial state depends on the props we pass to the component.
* As this is a static method we can’t make use of **this key word** we can’t call **this.setState()** instead you can simply an object that represent the new state of the component.
* Do not cause side effects, Ex: HTTP requests.

1. **Render()**

* This is required method in the class component.
* We simply read props & state and return JSX which describe the UI.
* This is a pure function it should render for the give props and state.
* Don’t change state or interact with DOM or make ajax calls.
* Right after parent **render()** Children components lifecycles method are also executed.

1. **componentDidMount()**

* This is invoked immediately after all its child components has rendered to DOM.
* Cause side effects. Ex: Interacting with DOM or perform any ajax or network calls to load data.

**Updating Lifecycle Methods**

1. **Static getDerivedStateFromProps(props, State)**

* This method is called every time component is re-rendered.
* It has to return null or object that represent the update state.